Retrofitting Objects

Robert E. Filman
IntelliCorp
1975 El Camino Real West
Mountain View, California 94025

Abstract

I present the results of an experiment in retrofitting objects into an existing system. I describe a technology, based on automatic redefinition of existing functions, that allowed alternative implementations of the fundamental data types of the KEE™ knowledge-based system building tool. This technology is applicable in environments where the system's procedures can be subject to programmatic manipulation. It allows the retrofitting of objects into the implementations of other existing systems. The experience of retrofitting objects into KEE provides insight into the issues of the interaction of semantic classes and data representation and granularity in object-based systems.[1]

[1] This research was supported by the Defense Advanced Research Project Agency under Contract F30602-85-C-0065. The views and conclusions reported here are those of the author and should not be construed as representing the official position or policy of DARPA, the U. S. government, or IntelliCorp.

1. Motivation

The KEE system is a tool for building knowledge-based systems [1]. KEE integrates several AI problem-solving paradigms, including frames, inheritance, production rules, access-oriented programming (demons), object-oriented programming, multiple-worlds, and truth maintenance, with facilities for querying, altering, and displaying the resulting structures.[2] The primary representational object in KEE is the *unit*. For example, we could represent Clyde the elephant as a `Clyde` unit. Units exist in a (multiple-parent) hierarchy, with both member and subclass links. For example, `Clyde` might be a member of the classes of `Indian.Elephants` and `Circus.Elephants`, themselves subclasses of `Elephants`. Relations are expressed as *slots* on units. For example, if the `Color` slot on `Clyde` has value `Pink`, then (one of) Clyde's colors is pink. KEE has two kinds of slots, *own* slots that describe properties of the object itself, and *member* slots for the properties of a class. Thus, `Color` on

[2] While there are several general insights gained from this research, the fiber of the work is heavily interwoven with its environment—the KEE system. KEE has been both the target and the tool; it will be difficult to understand this presentation without a few KEE concepts. The reader will therefore forgive me if I begin with a short overview of KEE. We should also note that the comments in this document about the implementation of the KEE system refer to version 2.1 of that system; some of the results of this research have been incorporated in the KEE system 3.0 release. KEE is a trademark of IntelliCorp.

Clyde is an own slot, while `Color` on Elephants, a member slot. Member slots *inherit* to member slots in subclasses and to own slots in elements. The particular form of the inheritance is controlled by the *inheritance role* of the slot (one of the *facets* that can be used to annotate slots). One of the representational tricks that's done with KEE is to place functions as the values of slots, allowing the user to perform object-oriented programming by *sending messages* to these units indexed by these slot names. The *method* inheritance role performs a simplified Flavors mixins [2], allowing assembly of methods from parts (*befores*, *afters* and *wrappers*).

Thus, the KEE system is a tool for, among other things, object-oriented programming. However, while KEE supports object-oriented programming, KEE itself is built using classical function calls and record-structured data.[3] In selecting these functions and data structures, the KEE system's designers optimized the system for representing, accessing, and dynamically updating a large variety of information about relatively few things. That is, users can dynamically create and delete slots, specify constraints on the number and type of values a slot can have, define new inheritance mechanisms, attach active values, etc. To support that functionality, units are non-trivial structures—they are large, require unique names, are sequentially stored in several places, and so forth. Effectively, the current KEE system implementation trades space (and, to a lesser extent, time) for generality and expressive ability.[4] This is not because the KEE system is poorly implemented—units are powerful, general structures that perform their tasks efficiently. But when a particular kind of unit always has a specific set of slots, or the values of a slot are never inherited, coerced, or mutilated by demons, the user is paying a significant storage and execution-time penalty for using the KEE system.

The goal of this work was to ease these limitations. Ideally, we would like to be able to treat any Lisp object as a unit, and to have the KEE system functions that manipulate units and slots deal with these new types of units. That is, we'd like a way to tell the KEE system that "user data type x," "integers between 100 and 1000," "strings," "lists whose `car` is the atom **banana**" and "files" are each possible data structures for units. Of course, we also have to (and want to) tell the system what it's supposed to do when someone asks for the value of the `fruit` slot of the string-unit `"Lisa loves apple juice."` Accomplishing these goals has the further by-product that we may be able to impose the KEE system on already existing programs—for example, adding KEE to an existing CAD system and then using it to reason about the devices being created.

In the above discussion, we have glibly spoken of treating arbitrary data structures as "units." But like most systems, KEE grew like Topsy, without formal specifications. Thus, when we ask, "What exactly is a unit?" we conclude that a unit is something that does unit-like things when the unit functions are applied to it. For example, after doing (`put.values unit slot values`)[5] the result of (`get.values unit slot`) is `values`. Lacking a more formal specification, we assert that the unit functions are the (approximately) 135 functions on units and slots defined in the KEE system users manual. We call the original KEE system implementation of units and slots *classical*; any ersatz implementation is *virtual*. A particular instance of a virtual unit is a member of some *virtual-unit type*. We use the term *shape* as a shorthand for virtual-unit type, attempting to convey the mapping to the different arrangements of machine storage used to implement different virtual-unit types. In general, a programmer wants to define a shape, and then to create instances that have that shape.

Now, if we really want units that behave just the way classical units do, we should have abandoned this project. The original designers of the KEE system did a good job of implementing the system core. The best way of achieving the full KEE system functionality with a reasonable time efficiency is to do it the way it was done. What we want from virtual units is a means of sacrificing functionality in return for improved time and/or space efficiency. For example, by decreeing that virtual units cannot be used to represent classes, we can build faster value storing functions—faster because they do not have to inherit values. By denying a particular shape user-settable facets, we can save storage. By declaring

[3] This being a consequence of the technology available at the time the KEE system was originally built and the necessity of quickly developing a fast implementation.

[4] For example, a unit with a single parent and four local single-valued slots requires 135 Lisp cells in the KEE system version 2.1. This is really only five cells of information.

[5] For the reader unfamiliar with the KEE system terminology, the Appendix is a glossary of the KEE system functions we mention in this paper.

that a shape has a specific set of slots, we can avoid dynamic slot allocation and search. Thus, we expect that most shapes will not have the full functionality of units—only the functionality important to the task they are to perform. Correspondingly, we are entitled to create shapes that behave differently than the classical shape—for example, a read-only unit that ignores calls to `put.values`, an unaccountable unit that does not record who modified it last, a logging unit that remembers everyone who accesses its slots, or an active database unit that responds to `get.value` calls by interrogating a database server. Thus, the same technology that enables our abbreviating or omitting functionality can be used to enhance or extend functionality.

Having asserted that a unit is something that does unit-like things, we ought to overview what (some of) those things are. A unit is primarily a storage structure. It stores four kinds of information: slot values; slot facets; unit and slot properties; and miscellaneous unit information, such as the unit name, inheritance links, knowledge-base, and creation time. (We call this last class of information *tags*.) There are functions for creating, storing, retrieving, and deleting each kind of information; often the modification of one kind of information can affect another. [For example, updating a slot changes the modification date. Semantically, checking the value of a slot involves retrieving the values of the same slot from the unit's parents (though classical KEE optimizes this action by updating values in the children when a parent changes).] Units can be tested for equality, dynamically created and destroyed, and saved on and retrieved from long-term storage. Units can also be displayed using the KEE interface, created and modified by queries and assertions, and examined in the forward and backward chainers. Thus, units live in a rich soup—many utilities already know how to deal with units; we want alternative implementations to take advantage of these utilities.

The remainder of this paper is devoted to describing how to retrofit objects into Lisp systems. I describe both a technology of object retrofitting that is applicable in many programming domains and the specific implementation of that technology in the KEE system. I have implemented that technology in the form of a prototype virtual-units KEE system and have built a prototype demonstration system using that technology. (This demo shows a system of 3200 virtual units simulating Conway's game of Life [3].) In this paper, references to the prototype and demonstration systems refer to these systems.

2. The application and construction problems

The above problem description demands, in essence, that we retro-fit object-oriented programming into the implementation of the core of the KEE system. The three primary elements of an object-oriented programming system are objects, messages, and handlers. *Objects* are the primitive program elements—the atoms of programming chemistry. The behavior of an object is induced by sending that object a *message*. Pragmatically, each behavior is implemented by running the appropriate *handler* (a program, or, in Lisp terms, a function) when an object receives a message. That is, message reception and function application are isomorphic. Thus, if the function **get.handler**(*object*, *msg-type*) determines the handler for *object* when sent a message of type *msg-type*, what an object-oriented programming system is really doing is

apply(**get.handler**(*object*,
message-type(*message*)),
(*object* . **args**(*message*))).

Thus, *Sending a message* is syntactic sugar for *applying a function*. What an object-oriented programming system does is simplify specifying *which* function ought to be applied in a given situation.

The task of building an object-oriented programming system has two primary subtasks. The first of these, the *application* subtask, involves arranging one's system so that the above application actually takes place. That is, if H is the handler for message M on object O, we want M messages to O to invoke H.

The careful reader has noticed that object-oriented programming is not magic—it is not providing us with any computational power beyond our original mechanisms. We could build the function corresponding to the universal interpretation of message m as a giant series of **if** ... **then** ... **else if** ... clauses. That is, if the object is in the class handled by handler h_1, then apply h_1, otherwise if it is in the class handled by h_2, and so forth. Object-oriented programming gives us a *nicer syntax* for saying these things. It is allowing us to define these handlers incrementally and textually local to the object class, not the message type (and, in implementation terms, may give us a more efficient indexing algorithm). Thus, the second subtask of the object-oriented programming system implementation task is building mechanisms to help the user construct and associate handlers with messages and objects. We call this the *construction* task.

So the task of implementing virtual units in the KEE system boils down to doing three things: (1) identifying the appropriate places to insert virtual units into the existing system, (2) changing the mechanisms of the KEE system to recognize when we've got a virtual unit and apply the appropriate handler (the application task), and (3) building tools to help the user develop virtual unit types (the construction task). I consider these in the next three sections.

3. Inserting virtual units into the core

One way to implement virtual units would be to reimplement the KEE system core, changing the unit accessing and modifying functions to recognize alternative unit implementations. Unfortunately, that approach has several drawbacks: (1) it runs the risk of dramatically slowing KEE, (2) it requires changing code in too many places, (3) it is a good way to infest the system with bugs, (4) it provides little positive guidance to the builder on to how to actually create a new kind of virtual unit, (5) it leaves unclear the semantic effect of changing parts of the underlying data structures, and (6) it requires recompiling the system to introduce another unit structure. These problems, particularly the fourth and sixth, imply that it was best to do this work on the "surface" of the KEE system's core. Let us examine that layer, and other layers, in greater detail.

3.1. Layers

The basic unit accessing and modifying functions form an architectural *layer*. The user-level functions (the 135 functions in the KEE system manual that we alluded to earlier) also form a layer. In general, it is an established design principle to build systems in layers, with a well-defined interface between layers. For example, classical hardware architecture provides a microcode machine to the microcode writer, microcode to the machine instruction writer, machine instructions to the compiler writer, and a programming language to the application writer. The virtual units project required identifying a layer in the software architecture of KEE, inserting at that layer the choice of the virtual alternative, and (for each shape) instantiating that layer. We call the 135 user-manual functions which are to respond to virtual units the *user layer*.

Unfortunately, the user layer is too big. We do not want to require an implementor of a shape to have to define 135 functions. These tend to be complicated functions, as they embody the full KEE system core functionality. On the other hand, while we don't want to require the implementor of a shape to have to define 135 functions, we don't want to preclude his having a particular special implementation of, say, `put.values`. That is, we want to *require* the definition of as few functions as possible, but to *allow* the redefinition of as many as desired.

Our resolution of this problem centers on selecting a core set of functions, the *virtual layer*. With these functions, we can implement the functions on the user layer (*Figure 1*). In the prototype system, this layer has 34 simple functions. Typical functions at the virtual layer include creating a slot in a unit, retrieving the local value of a slot, and storing a new local value. Thus, in addition to being smaller than the user layer, the functions in the virtual layer are typically semantically simpler.

We chose the functions in the virtual layer to allow the programmatic expression of the functionality of the user layer. For example, the user-layer function `get.value` can, independently of the implementation of its operands, be defined as the `car` of the `get.values` on the same operands (preserving its original semantics). Similarly, the user-layer function `add.value`, parameterized (roughly) over `unit`, `slot`, and `new.value`, can be implemented as (1) creating `slot` if it does not already exist, (2) retrieving the datatype of `slot`, (3) coercing the new value according to the datatype, (4) retrieving the `old local values` of `slot`, and (5) if `new.value` is not one of them, storing the addition of `new.value` to `old local values` in `slot`, (6) retrieving the values of the parents of `slot`, (7) retrieving the inheritance role of `slot`, (8) combining the new values with the parent values, (9) installing the result as the derived value of `slot`, (10) determining the appropriate active values, and (11) running them. Each of these steps is a programmatic combination of virtual-layer elements. The key point is that the user-layer functions can be (programmatically) built from the virtual layer.

4. The application task

Functions become sensitive to virtual units by being "advised" to check their arguments before executing their normal behavior. Advising a function is redefining it by wrapping additional code around the original definition of the function. This works in the Lisp environment because calls to function F are routed through the function cell of F. (We could perform

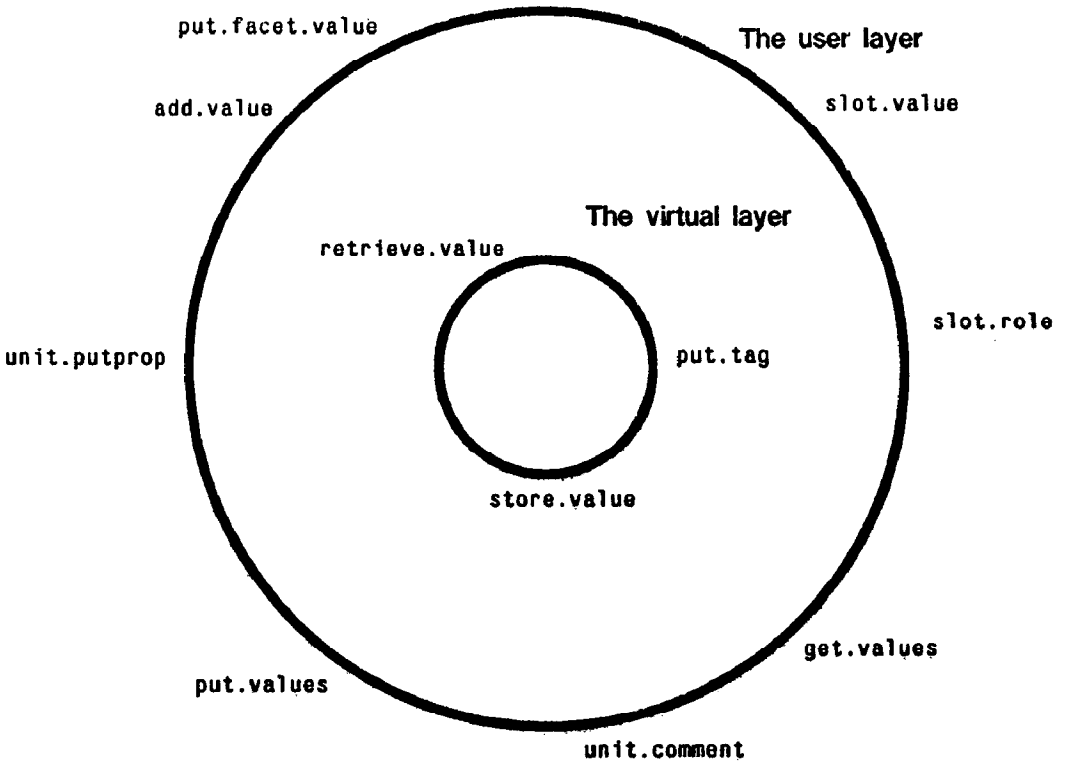

Figure 1: The virtual and user layers

the same trick in compilation-based environments if we had access to the source code and convenient systems for automatically parsing and modifying it.) In our implementation, this advice is generated automatically when the user specifies that a particular function is to be sensitive to virtual units.

The application process requires discovering the appropriate handler for a message and then invoking that handler. If we are to find the right handler for something, we need some characteristic that distinguishes the different classes of things. These are the shapes. Each virtual unit (that is, anything that is to be treated as a unit) must belong to a shape. It must be possible to determine (constructively) the shape of a given virtual unit. That is, we need a function that, given a candidate unit, returns its shape if it is a virtual unit (and `nil` if it is not). To obtain adequate system performance, this function must be fast. For example, a typical use of virtual units might include only shapes that are user data types. In many Lisp systems, the data type of an object is easy to determine and is represented by some symbol. An object whose data type symbol has a particular property could then be recognized as an element of that shape; the value of that property could be an index of handlers. Alternatively, we could restrict virtual units to array-shapes (like Flavors), keeping the handler table in the first element of the array.

When an advised function is called with a virtual unit, it should obtain from that unit's shape the *handler* for that function. The handler is the definition of that function for that type of unit. We run that handler on the virtual unit and the remaining arguments. That is, to invoke the function `(f u ...)` on the virtual unit u, we want to:

```
(apply* (lookup (shape u) f) u ...)
```

This expression is, of course, isomorphic with the expression that defines an object-oriented programming system. We call the result of `shape` a *representation structure*.

In the prototype implementation, we have two different types of objects as possible values of function `shape`: classical KEE units and hash arrays. In the first case, `lookup` does a `get.value` with `f` as its slot; in the second, `lookup` does a `gethash`.

The following code summarizes the change in the behavior of advised functions. To sensitize a function `(f u a b c ...)` to its first argument, `u`, as a virtual unit, we redefine it as

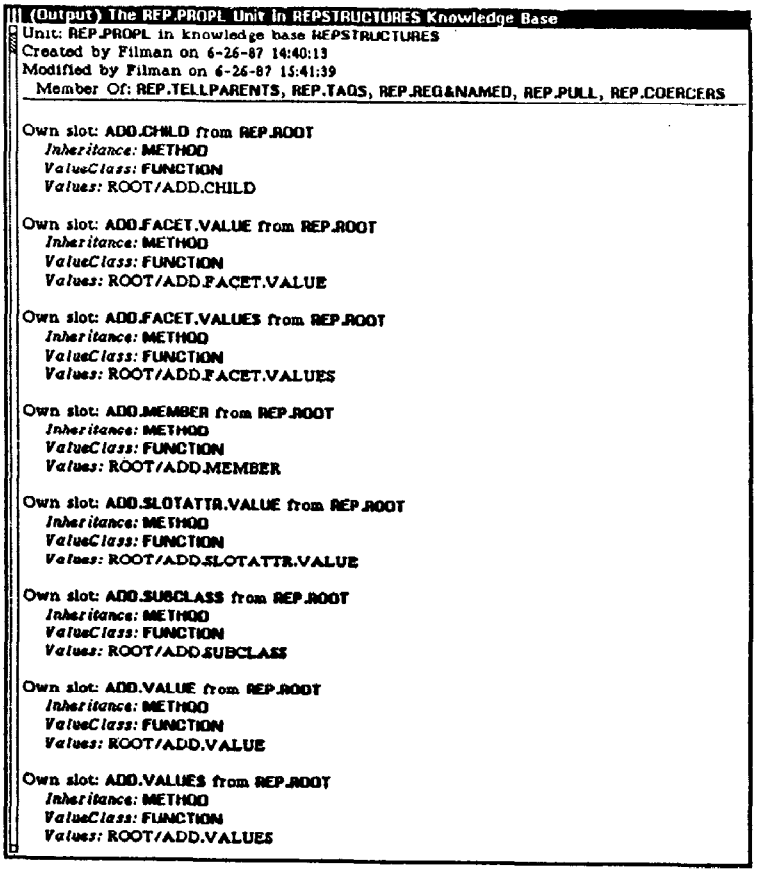

Figure 2: Unit display of part of REP.PROPL

```
(if (classical.unitp u)
 then (f' u a b c ...)
 else
   (let ((u' (or (unitreference* u) u)))
     (if (classical.unitp u')
      then (f' u' a b c ...)
      else
        (let ((vrep (shape u')))
          (if vrep
           then
             (let ((handler
                     (lookup vrep f)))
               (if handler
                then
                 (apply* handler u'
                         a b c ...)
                else
                 (error "Unknown handler")))
           else (f' u a b c ...))))))
```

where `classical.unitp` tests if a unit is a classical unit, and `f'` is the original definition of `f`. That is, first we check to see if we've been given a classical unit data type. If so, we execute the original definition of the function. Next we consider if we've been given the name of a unit. We search for a unit by that name; if we find a non-classical instance, we apply the handler of that unit's shape and function to the appropriate arguments. Otherwise, we execute the original function. (This particular definition is skewed towards minimizing interference with the classical KEE system—we check twice for classical representations before considering virtual alternatives. In a system dominated by virtual units, the first of these checks could be eliminated. Also note that, in contrast to most other systems, this definition is complicated by KEE provision of both pointers to unit structures and separate unit names, either of which can be used to reference a unit.)

In practice, we have defined a series of functions that add the appropriate advice to KEE system core functions. Different functions are used for core functions over one, two and three "unit" arguments, and for core functions that reference slots. Variants and parameterizations of these functions allow for different orders of parameters, alternative handlers, and for the computation of fix-up functions after calling an alternative handler (for example, redefining `get.value` to be `(car get.values)`). These tech-

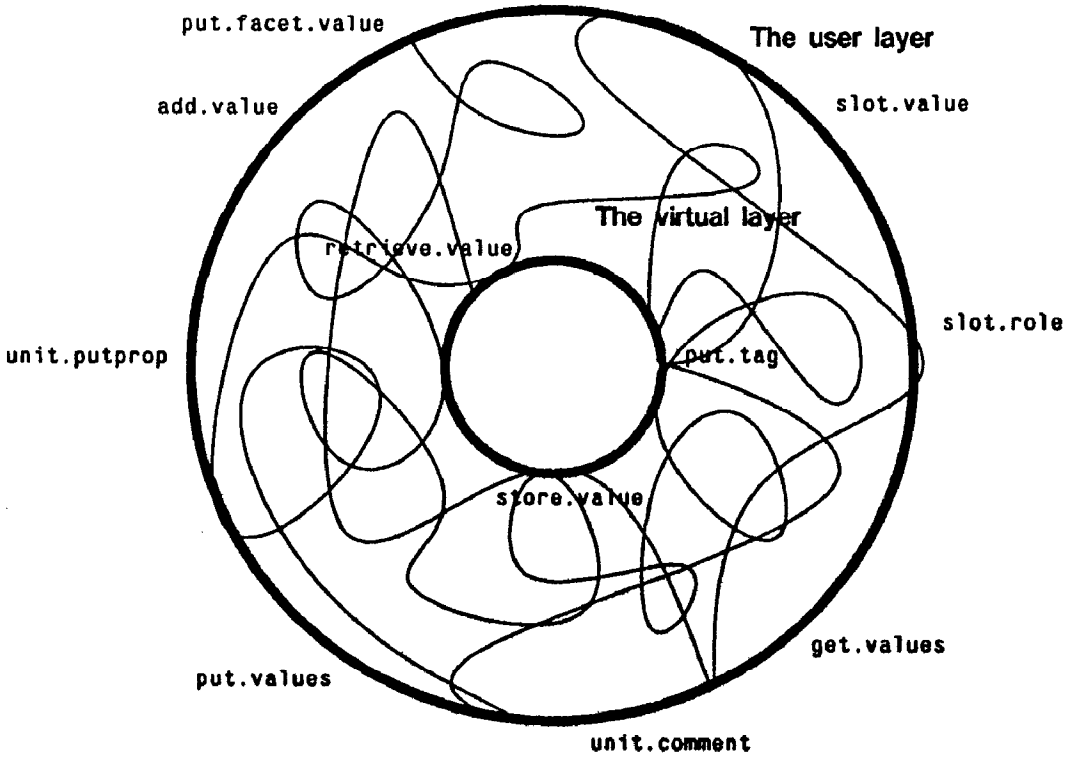

Figure 3: Spaghetti between layers

niques fail when the underlying functions exhibit too much syntactic variation or are themselves macros for compiler optimization. In the KEE system, about a dozen functions need individual treatment, such as the functions for returning a unit or slot from unit and slot names and the functions for creating a unit (as there is no virtual unit object on which to index before the creation).

5. The construction task

For the purposes of ease of program development, automatic inheritance and method combination, we choose to represent collections of handlers as units. The idea is that the representation structure of any shape is a unit. Each advised function is the name of an own slot in that unit; the handler for that advised function is the value of that slot. *Figure 2* shows the first part of the unit display of the representation structure unit `REP.PROPL` for the shape PROPL units. The actual unit has about 180 slots. (PROPL units store their data in "property-list format," and are uninteresting except for their complete functionality. The additional facets in the unit display are directives to the automatic advice mechanism.) Thus, doing an `add.child` on a unit of shape PROPL retrieves the function `root/add.child` from the `add.child` slot of `REP.PROPL` and applies that function to the original arguments.

The advantage of representing representation structures as KEE system units is that we can use method inheritance to obtain default values for handlers. (As discussed below, we can also use method combination to assemble handlers from components.) We provide default implementations of the user-layer functions in terms of the virtual-layer functions. We call the implementation of a higher-layer function in terms of a lower-layer ones *spaghetti*, as higher-layer functions take a twisting and tangled path down to the lower-layer functions (*Figure 3*). The implementor of a shape who wishes a specialized version of a user-layer function can defeat this default. Using the KEE system's standard inheritance mechanisms (typically method inheritance for functions) we obtain both default behaviors and the ability to override defaults.

For each advised function, the implementor of a shape can (1) inherit the default action, (2) specify a particular action, or (3) simply leave the action empty. That is, we have virtual units because we do not need all the functionality of classical units; one way of simplifying functionality is to omit functionality. For

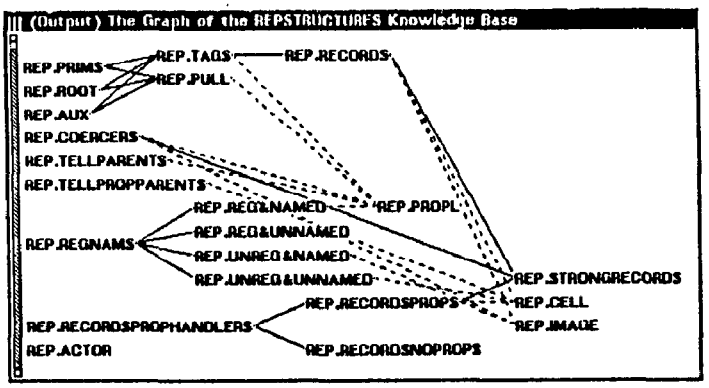

Figure 4: The REP.STRUCTURES knowledge base

example, we expect that shape implementors will often create shapes that lack unit names, are not known to their knowledge base or parents, not writable to permanent storage, lack facets, or do not coerce values before putting them in slots.

Figure 4 shows the REPSTRUCTURES knowledge base from the prototype demonstration system. Solid lines indicate subclass relations; dashed, set membership. It has representation structures for four shapes: REP.PROPL, REP.IMAGE, REP.CELL and REP.ACTOR. REP.PROPL units are four-element lists: a special symbol (**$$$PropL$$$**), and lists of slots, tags, and properties, each stored as a property list. REP.IMAGE and REP.CELL are compact record-structured representations of units with known slot structures. These representations were generated by calling a representation-structure generating function; some of the handlers were then improved by substituting optimized functions in particular slots. REP.ACTOR units are storage for simple active values (demons). They take up five cells, have room for each of the four KEE 2.1 system active-value slots, and have only the minimal handlers to establish values and invoke demons.

At the top of the REPSTRUCTURES knowledge base are the class units REP.PRIMS, REP.ROOT and REP.AUX. Unit REP.PRIMS has member slots for each function on the virtual layer; REP.ROOT, each function on the user layer. Thus, there are 34 slots in REP.PRIMS and 135 in REP.ROOT. Unit REP.AUX contains about a dozen "auxiliary" functions that express recurring concepts in the space between the user layer and the virtual layer—for example, the concept of the "active value units interested in this slot." In general, the default values in REP.ROOT are programmatic combinations of the functions in these three units. That is, functions such as `get.values` and `put.facet.value` are expressed in terms of the available primitives. The default values in REP.PRIMS are empty—these are the foundation on which other data types are built; we can say little about them *a priori*. (Slots in REP.AUX are like REP.ROOT; they differ in that they are functionality needed by shape builders, not KEE system users.)

We expect that, except for specialized shapes, all shapes would be children of these three. Thus, units such as REP.PROPL have slots for each virtual-layer function, each primitive function, and each auxiliary function. The values of the virtual-layer and auxiliary functions are typically inherited, while the primitive functions are more locally defined.

The middle of the unit structure of the knowledge base is dominated by units such as REP.PULL and REP.TAGS. The idea here is that certain implementation decisions imply that a class of virtual functions can support related handlers. For example, REP.PULL embodies the idea of "pull inheritance," described below. It defines handlers for intermediate value storage and retrieval functions such as `retrieve.data`, `retrieve.facet.data`, and `store.data`. REP.TAGS compresses the idea of additional information (the tags) being stored and retrieved through one uniform mechanism. That is, the developer of a shape provides primitives for storing and retrieving a $\langle tag, value \rangle$ pair, and the tags structure defines functions such as `unit.comment` as retrieving the $\langle \texttt{comment}, x \rangle$ value.

5.1. Programming with components

The units REP.COERCERS, REP.TELLPROPPARENTS, REP.TELLPARENTS, and the subclasses of REP.REGNAMS and REP.RECORDSPROPHANDLERS store information for *component programming*. The idea here is similar to that available with mixins in the flavors packages of Zeta-lisp [2]. Often, functionality can be expressed

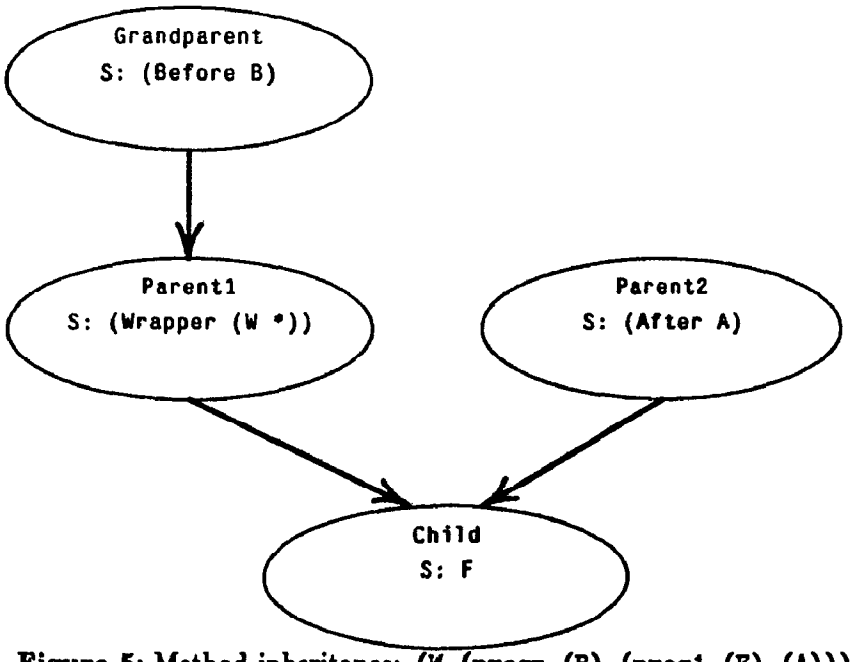

Figure 5: Method inheritance: (W (progn (B) (prog1 (F) (A))))

as a collection of actions to be taken (grains). For example, creating a unit with name x involves not only allocating the storage for that unit, but also interning x in the system's oblist mechanism and informing the current knowledge base of the new unit (registering it). These last two are independent of the shape of the newly created unit, and independent of each other. The creator of a particular shape might not want the conventional interning and registration actions. That is, units in a particular shape might not have names, or might always be put in a particular knowledge base independent of the current knowledge base. Nevertheless, the naming and registering activities need to be coordinated over different functions—the action taken on creating a unit implies particular corresponding actions on transferring that unit to a different knowledge base, and renaming, copying, or deleting it.

The method inheritance mechanism in the KEE system allows us to take advantage of the natural grain of programs. This mechanism allows the user to specify *mixins*, "before," "after," and "wrapper" actions to be combined with a major functionality. *Figure 5* shows a unit hierarchy with the local values for several units for slot s. This slot inherits with method inheritance. The derived value of this slot is `(W (progn (B) (prog1 (F) (A))))`.

Some activities can be divided into grains, and some of these grains have alternative behaviors for different shapes. We can take advantage of this graininess in the structure of our representation knowledge base. In particular, we implement the related alternatives as mixins on slots of representation units. Any shape that is to have such behavior can simply be made a member of such a class unit. For example, in the REP-STRUCTURES knowledge base, units of shape REP.PROPL are installed on their parent's list of children, coerce new values, and registered and named by making REP.PROPL a member of the classes REP.TELLPARENTS, REP.COERCERS, and REP.REG&NAMED. If we did not want REP.PROPL units to coerce their values, we would simply omit the link to REP.COERCERS.

5.2. Automatic generation of shapes

The unit REP.RECORDS represents a schema of shapes. Units of this form are a record structure with a predefined set of slots. We have a function, MAKE.VIRTREC, parameterized by attributes such as (1) the name of the new shape, (2) the names and valueclasses (type constraints for valid values) of the particular slots of this shape, (3) default classes and prototypes for newly created units, (4) whether additional slots, facets, or unit or slot properties are to be allowed, and (5) whether units of this shape are to coerce their values, inform their parents, be stored in their parents, and/or inform their knowledge base. Execution of MAKE.VIRTREC creates a shape (and its associated unit) of the given name and with the specified properties. In the demonstration knowledge base, REP.CELL and REP.IMAGE are precisely this kind

of representation. Component functions, such as value coercion and naming, are achieved by making (or failing to make) the newly created representation structure a child of the appropriate component unit. MAKE.VIRTREC writes functions specific to other parts of the parameterization for the primitives required for that particular shape; the other primitives and composite functions are inherited from REP.RECORDS. For example, if we create a record-structure shape with fields A, B, and C, MAKE.VIRTREC defines field-accessing and field-storing functions for that shape that include **case** statements over the names A, B, and C.

6. Run-time considerations

6.1. Untangling the spaghetti

The spaghetti mechanism (defining high-layer functions in terms of combinations of lower-level ones) in many ways reflects standard programming practice. However, the virtual unit technique requires re-deriving the shape of a particular unit and searching for its handler repeatedly through the spaghetti tangle. This re-derivation and search can, in many cases, be compiled-out. That is, we can (and have) automatically transformed the handlers of particular shapes into macros that recognize that their unit is of the given shape and compile precisely the required routine.

6.2. Development and execution environments

This section has described a *development* environment for building virtual units. At run-time, it is only important that the values of the handlers be present, not the structures used to build them. Thus, the REPSTRUCTURES knowledge base is not needed at execution-time—only a structure to map between the external function name ("**get.values**") and the handler for that function for a particular shape ("**propl/get.values**"). While this structure can be a unit, we have also implemented it as a hash table and as a record of pre-defined field names.

7. Limitations of this approach

This approach has several limitations. Some of these are caused by our desire not to completely imitate the functionality of classical KEE. The most prominent is the timing of inheritance. The KEE system modifies the children of a unit when a member slot is changed. We call this *push inheritance*, because values are "pushed" from parents to children. Thus, when the value in a parent changes, demons attached to the children can awake and make their effects visible. In a large, virtual-unit–based system, we can expect that most classes will not have an explicit list of their children. This may be because: (1) there are too many children to list, (2) the children exist only implicitly in some database or on permanent storage, (3) the children form an infinite set (*e.g.*, "the integers"), (4) the children exist only implicitly as the result of some computation, and (5) for garbage collection, storage space, or security reasons, extra pointers to the virtual units are inappropriate. Obviously, one cannot do push inheritance to unknown children. Instead, any inheritance of values to such units must take place at access time. We call such inheritance *pull inheritance*. Ordinarily, the time of inheritance would be invisible to a user—the most straightforward (*i.e.*, side-effect free) semantic definition of inheritance cannot distinguish between systems implemented with push and pull inheritance. However, the KEE system's active value mechanism makes the internal workings explicit. By appropriately using (or misusing) active values, a user can obtain a great deal of information about the ordering of the KEE system's internal operations.

Who points to a unit? The KEE system keeps a pointer to a unit on the property list of its name (the KEE system's implementation of an oblist), in the unit **entities**, in its parents and children, in the unit's knowledge base, in the value part of anything it is a value of, and on the property list of some of its slotnames (for pattern-directed queries). The implementor of a shape can choose which of these to perpetuate. In particular, it may prove worthwhile to reimplement the search mechanism for those slots for which linear search is inappropriate.

The virtual unit system described above, where higher layers are implemented in terms of lower ones, suffers in that the system repeatedly determines the shape of a unit as it unravels the spaghetti. This can be alleviated by (1) compiling-out the lookups, and (2) restructuring the KEE system to reduce the type-dependent distance between the user layer and the virtual layer. That is, if the system definition of **get.value** is **(car get.values)**, there is no cycling penalty for using **get.value**.

In the prototype virtual unit implementation, I made several assumptions that serve as *a priori* limitations on the functionality of virtual units. I assumed that (1) virtual units cannot be classes (they cannot have subclasses or members, or member slots), (2) virtual unit slots are multiple-valued, (3) active values are not necessarily run at the same time and in the same order as classical units, and (4) slots are

not unique data structures. These assumptions are not inherent in the virtual unit mechanism, but were chosen for expediency.

8. Advantages of this approach

Having listed our faults, we are entitled to mention a few of our virtues. Of course, there are clearly many performance advantages in having virtual units. In this section, we are concerned with the advantages that this approach provides, in contrast with other possible implementation alternatives.

1. **Class $\neq$ shape.** We have separated the semantic class of objects from their implementation type. We do not have to implement all the instance units of a particular class in a particular format, or to assume that all units of a particular format belong to a particular class. Traditional implementations of objects (for example, [4]) confuse the semantic basis of an object with its implementation. (Of course, it takes having a rich, pre-existing semantic environment like a knowledge representation system before one even has much of a semantic basis available). We allow a given virtual unit to be in several classes, and to change classes dynamically.

2. **Building a shape is not wizardry.** We make it clear what the implementor of a shape must provide, and give him an explicit (and visible) place to put it. The operations tie directly to the semantics of the KEE system; no knowledge of magic flags or bits are required (though an implementor can include her own set of magic flags). Building a shape is straightforward enough that it can be done programmatically.

3. **Implementation can be inserted at any layer.** Since the virtual handler is consulted for all advised functions, the implementor can insert an optimized version of any advised function for any shape. He can also advise (almost) any function. However, because of the `lookup` step in the virtual advice, we cannot optimize any operation to be faster than a symbol to value translation (typically `gethash`, classical `get.value`, or `getprop`), though this translation may be possible, in many cases, at compile time.

4. **Implementations can be built from components.** By identifying the grains of an action, we can sequester these grains into components. We can selectively use these components to build gross actions.

9. Status of the Implementation

As part of our research, we have built a prototype implementation of virtual units. This prototype includes the full user-layer functionality. We implemented several shapes, and created a function that generates shapes automatically. IntelliCorp's Product Development and Engineering Department has taken many of these ideas and incorporated them in the internals of the latest release of the KEE system, KEE 3.0.

10. Comparison with other work

Traditionally, if one wants to do object-oriented programming, one programs in an object-oriented system. One does not build one up around oneself. The other interesting exception to this rule is the Portable Common LOOPS system from Xerox [5]. That system also takes advantage of functional redefinition to implement objects in an existing (Lisp) environment. Common LOOPS provides the ability to sensitize any function to shapes of different units on several arguments. This implies that they have developed a more sophisticated set of program-modification functions than has been required to insert virtual units in the KEE system.

11. Summary

While we have described the virtual units project in terms of changing the KEE system, the techniques involved are applicable on a much wider scale. The moral is that if one is in an environment where the definitions of procedures can be programmatically varied, it is possible to build a set of tools that allow object-oriented programming—even in an already constructed system. If the semantics of that system allow programmatic expression in terms of a simpler, lower layer, then the object retro-fitting can be much more straightforward. And if one can recognize the underlying conceptual grains, the object retro-fitting can combine these grains as appropriate.

Acknowledgments

I would like to thank Greg Clemenson, Bill Faught, Richard Fikes, Bob Nado, and David Silverman for insightful discussions, thoughtful commentary on drafts of this paper, and guidance in fighting my way through the KEE core.

References

[1] Fikes, R., and Kehler, T., "The role of frame-based representation in reasoning," *CACM, vol. 28*, no. 9, September 1985, pp. 904–920.

[2] "User's Guide to Symbolics Computers, Volume 4," Symbolics Corporation, 1985, pp. 111–146.

[3] Gardner, M., *Wheels, Life and Other Mathematical Amusements*, New York: Freeman, 1983, pp. 214–257.

[4] Goldberg, A. and Robson, D., *SMALLTALK-80: The Language and Its Implementation*, Reading, Massachusetts: Addison-Wesley, 1983.

[5] Bobrow, D., Kahn, K., Kiczals, G., Masinter, L., Stefik, M., and Zdybel, F., "Common LOOPS: Merging Lisp and object-oriented programming," in OOPSLA '86: Object-Oriented Programming Systems, Languages, and Applications, Portland, Oregon, September 1986, pp. 17–29.

Appendix: Glossary of KEE system functions

Table 1 describes the KEE system functions mentioned in this paper.

Function (arguments)	Description
`add.child (unit, parent, link-type)`	Makes `unit` a member or subclass of `parent`, depending on the value of `link-type`.
`add.value (unit, slot, new.value)`	Adds `new.value` to the set of values of `slot` of `unit`.
`get.values (unit, slot)`	Returns (as a list) the set of values of `slot` of `unit`.
`get.value (unit, slot)`	Returns one of the members of the set of values of that slot.
`put.values (unit, slot, new.values)`	Changes the values of `slot` of `unit` to be `new.values`.
`put.facet.values (unit, slot, facet, new.values)`	Changes the values in `facet` of `slot` of `unit` to be `new.values`.
`unitreference (name, kb)`	Looks for and returns the unit of name `name` in knowledge base `kb`.
`unit.comment (unit)`	Returns the comment (text description) for `unit`.

Table 1. Representive KEE system functions